Evolutionary Algorithms: Final report

Sander Prenen (r0701014)

December 16, 2020

1 Metadata

- **Group members during group phase:** Karel Everaert and Matthias Maeyens
- **Time spent on group phase:** 15 hours
- **Time spent on final code:** 79 hours (including hyperparameter search)
- **Time spent on final report:** 10 hours

2 Modifications since the group phase

2.1 Main improvements

Initialization of the population: A greedy heuristic was used to initialize the population in order to start from a fitter population than the random population used in the group phase.

Schemes: Different schemes were carried out for all the main operators in the evolutionary algorithm (selection, mutation, recombination, elimination).

Implementation of a local search operator: A local search operator is implemented to improve the candidate solutions and speed up the convergence.

Ability to 'restart' the algorithm: When the algorithm gets stuck in a minimum, it is able to restart itself with a new population and tries to converge to another (better) minimum.

Parallelization of the most expensive parts of the algorithm: Computationally expensive parts of the algorithm like the local search operator have been parallelized by using the `multiprocessing` library.

2.2 Issues resolved

Mutation: The mutation operator used in the group phase had little effect on larger problem sizes. A number of different mutation operators were implemented to cope with this problem.

Elimination scheme: The algorithm created in the group phase had the problem that the fittest candidate solution got mutated unintentionally, making divergence from the optimal tour possible. This issue was fixed by integrating an elitism elimination scheme to always keep the fittest individual in the next generation.

Diversity promotion: The implementation in the group phase suffered from a lack of diversity. This caused the algorithm to converge to local optima. This problem was solved by adding a diversity promotion scheme to the evolutionary algorithm.

Recombination operator: The recombination operator used in the group phase did not preserve the order of the nodes in the tour, but rather tried to preserve the position of the nodes. Several new recombination operators were created to deal with this problem.

3 Final design of the evolutionary algorithm

3.1 Representation

An individual is represented as a 1D `numpy` array with the order of the visited nodes. A `numpy` array is used rather than a plain python list, because of the more efficient manner in which these arrays get stored in memory and the vectorized operations that are available in the `numpy` library. A population is represented as a two dimensional array (each row is an individual). This choice was made for the same reasons as mentioned above. Since duplicate

individuals are allowed a set cannot be used. This type of collection is also unordered which would require more computations.

3.2 Initialization

The population is initialized by using a greedy algorithm. The algorithm starts in a node and picks the nearest not visited node as the next node. The starting points are decided at random. This is a variation on all nearest neighbors (ANN) as proposed by Kaabi and Harrath [1]. This heuristic method creates less diversity than random initialization. Therefore a diversity promotion scheme is implemented as described in section 3.8. The random initialization method is still available by setting the boolean value of `use_random_initialization` to true. This version explores more of the search space, but it will never reach the objective value of the heuristic solution for the bigger problems in the given time span.
When the algorithm gets trapped in a (local) optimum, it will restart itself by initializing a new random population. These individuals are created using the random nearest insertion heuristic [2] and can then recombine with the original individuals to escape the optimum. This heuristic uses a random sequence of nodes and inserts it into an existing sequence whilst keeping the total length minimal.

3.3 Selection operators

The following selection operators were tested: k-tournament selection, linear rank-based selection and roulette wheel selection with geometric decaying selection pressure. Some of these selection operators depend on parameters chosen by the user. For k-tournament this parameter is k, the number of individuals to be randomly selected for the tournament. The roulette wheel selection depends on the starting selection pressure and the decay for this selection pressure. The linear rank based selection does not need any parameters to be chosen.
The final version of the evolutionary algorithm uses the roulette wheel selection with geometric decay. By doing a hyperparameter search, as described in section 3.11, the values of the starting selection pressure and the decay are both set to 0.999. This value was set when the entire algorithm was finished and was based on the number of iterations that were typically needed in order to reach convergence. The generation of individuals in this operator is not done using inverse transform sampling, as this would require $\mathcal{O}(n)$ time per sample. Instead the Vose's alias method is used as explained by Schwarz [3, 4]. This method is able to sample in constant time, but uses $\mathcal{O}(n)$ time to initialize. This initialization can be reused to select all the individuals during the current generation.
A fitness proportionate selection scheme was briefly introduced but discarded due to the problems mentioned in the slides [5].

3.4 Mutation operators

The mutation operator that existed in the group phase, the sequence swap operator, was simplified to a simple swap operator that swaps two nodes at random. This operator had little effect on larger problem instances due to its proportionally small range. Therefore three different operators were introduced: reverse sequence mutation (RSM), partial shuffle mutation (PSM) and a hybrid of the two, called hybridizing PSM RSM mutation (HPRM) [6].
RSM chooses two nodes in the individual at random and reverses the sequence between the two nodes. PSM is similar but randomly shuffles the sequence between the nodes, thus introducing more randomness. HPRM is a combination of the two where the sequence between the nodes is reversed element-wise and shuffled after each reversion.
All the operators discussed cannot be controlled by a parameter to tune the amount of randomness they introduce. The final evolutionary algorithm uses RSM because it proved the best in the numerical experiments. The operator is applied with a probability α to an individual, with $\alpha = 0.2 * \frac{\textit{mean objective}}{\textit{mean objective} + \textit{best objective}}$. This idea is based on the idea from Kaabi and Harrath [1].

3.5 Recombination operators

The recombination operator implemented during the group phase, partially mapped crossover (PMX), was kept. Two other recombination operators called order crossover (OX) and sequential constructive crossover (SCX) proposed by Davis [7] and Ahmed [8] were implemented. All three operators start from the same principle, two crossover points are chosen at random in both parents. The nodes between these points are then copied to the offspring. How the rest of the nodes are copied differs between the operators. PMX tries to keep the positions of the nodes consistent by using cycles in the parents. OX copies the unused nodes from the parent of which the nodes between the crossover points have not been copied to the child. SCX searches in both parents to the next unused node and picks the one that is the nearest to the node previously inserted in the child. This operator thus only creates one child from two parents. Therefore double the number of recombinations are needed in order to generate the same number of offspring.
Since OX and SCX respect the relative order of nodes in the parents, these will probably perform better than PMX

on the asymmetric TSP. The experiments performed in section 3.11 also show this. The final algorithm uses OX since it had better results on bigger problem instances. This operator guarantees that if an edge between two nodes is present in both parents, it will definitely be copied to the offspring. If there is little overlap between the parents, then the operator will practically copy half of both parents to the children. Due to the complexity of recombination operators on permutations, operators with arity greater than two have not been considered. The operators themselves don't depend on any parameters. They just create the number of children specified in the `offsping_size` variable.

3.6 Elimination operators

The only elimination scheme implemented is $(\lambda + \mu)$-elimination, which was created in the group phase. Since it is not guaranteed that the offspring will be better than the previous generation, this is the best option. To make sure that the best objective value will not diverge, elitism is applied to the population after elimination. This is described in section 3.12.
The selection operators k-tournament and roulette wheel selection were also tested as elimination schemes, but the results of these proved worse than the $(\lambda + \mu)$-elimination. This was probably the case due to the time limit, because the selection operators had a higher diversity. This diversity caused that the algorithm was not able to converge as quickly.

3.7 Local search operators

Three different local search operators where considered. All three operators are based on a neighborhood structure. The first one is a 3-opt method where a tour is seen as three different arcs (A-B-C). If the alternative A-C-B is shorter then this tour is kept. The second operator is a 2-opt operator. This operator swaps two neighboring nodes in an individual if it is shorter than the other way around. The last operator reverses the individual as a whole if it is shorter. This operator was not implemented since it would not have a big effect because of the way the distance matrix is generated.
The final algorithm uses the 3-opt local search operator based on the one proposed by Kanellakis and Papadimitriou [9] and improved by Bentley [10] and Nagata and Soler [11]. Instead of splitting the tour in arcs in all possible places, a list of nearest neighbors is used for each node. Once a first splitting point is chosen, then the second point is limited by this list of nearest neighbors. This reduces the computations needed to perform the local search while still receiving a good improvement on the individual. To reduce the computations even further, the first splitting point is chosen at random. This makes that there is not always a better individual found using the operator, but if the individual makes it to the next generation then another random splitting point will be chosen. This approach proved experimentally better than doing a full 3-opt search on the individuals.
The only parameter that needs to be chosen is the number of nearest neighbors that is kept in the neighbor list. Nagata and Soler suggested to use 15. This is also the value used in this evolutionary algorithm.

3.8 Diversity promotion mechanisms

Two different diversity promotion schemes have been implemented. Fitness sharing in the elimination phase and the removal of duplicate individuals after elimination as proposed by Herrera-Poyatos and Herrera [12].
Fitness sharing is implemented as described in the slides of this course [5]. The distance function used is the number of edges that are different between two individuals. The computation of the distance function is very expensive and is therefore executed in parallel. This scheme needs a maximum distance σ that will be considered in order to apply the penalties.
The removal of duplicate individuals is based on the objective values. If the objective values are not equal then the individuals cannot be the same. This check is much easier than the distance computation in fitness sharing. If two or more individuals are the same, then one is kept and the others are replaced by a greedy randomized algorithm. The individual is initialized starting from a random node and the next chosen node is picked randomly from the set restricted candidates. These candidates are all nodes that are less than σ percent longer than the greedy option.
The final algorithm uses the greedy diversification option because it performed better on the larger problem instances. Computing the distances for all individuals proved to be too costly. The parameter σ used in the greedy diversification algorithm is set to 10 as suggested by the authors of the paper.

3.9 Stopping criterion

During the development phase, a combination of two criteria was used. The first one was a limit on the number of generations (500 in this case) and the second one was the number of generations in which the best solutions had not changed. Some trail-and-error showed that 30 was a good value for this. The former criterion was obsolete for the larger problems due to the expensive local search operator. Since the elitism scheme was used in the final version of the algorithm, the stopping criterion has been dropped in order to fully use the given time span.

```
def optimize(self, filename: str):
    """
    The main loop of the genetic algorithm.
    :param filename: The filename of the tour for which a candidate solution needs to be
                                                        found.
    """
    # Read the distance matrix from file
    data = np.loadtxt(filename, delimiter=",")
    self.tour_size = data.shape[0]
    self.raw_distance_matrix = RawArray(ctypes.c_double, self.tour_size * self.tour_size)

    self.distance_matrix = np.frombuffer(self.raw_distance_matrix, dtype=np.float64) \
        .reshape(self.tour_size, self.tour_size)
    np.copyto(self.distance_matrix, data)

    self.build_nearest_neighbor_list()
    self.init_dictionary()

    population = self.initialize_population()

    while not self.is_converged():
        offspring = self.recombination(population)
        mutated_population = self.mutation(population, offspring)

        optimized_population = self.local_search_parallel(mutated_population)

        population, scores = self.elimination(optimized_population)
        population, scores = self.eliminate_duplicate_individuals(population, scores)
        population, scores = self.elitism(population, scores)

        if self.same_best_objective % 15 == 0 and self.same_best_objective != 0:
            if self.same_best_objective >= 40:
                break
            self.selection_pressure = 0.8
            population = self.random_nearest_insertion()
            population[0] = self.best_solution
            scores = self.length(population)

        if self.mean_objective != np.inf:
            self.alpha = (0.2 * self.mean_objective / (self.best_objective + self.
                                                       mean_objective))

        self.update_scores(population[0], scores)

        time_left = self.reporter.report(
            self.mean_objective, self.best_objective, self.best_solution
        )
        self.time_left = time_left
        # if time_left < 0:
        #     break
```

Figure 1: The main loop of the evolutionary algorithm

3.10 The main loop

Before the main loop starts, some parameters that depend on the tour size are initialized. The offspring is generated in the `recombination` function. In there the individuals are selected with replacement in parallel. Afterwards all parents and children are supplied to the `mutation` function. This function applies mutation with a given probability α specified in section 3.4. Then the local search operator is applied on half of these individuals. Elimination and diversification is then applied to the optimized population. The next part is 'restarting' the algorithm if there is a risk of settling in a local minimum. Finally the results of this generation are sent to the reporter and the loop is restarted. The main loop can be seen in Figure 1.

3.11 Parameter selection

Tuning all parameters of an evolutionary algorithm is huge problem on its own. Therefore not all possible combinations of the parameters and operators were tested. The tuning process was split in multiple parts in order to decrease the time needed. The first part of this process was deciding the mutation and recombination operators out of the possibilities described in section 3. All the different combinations of mutation and recombination operators were tested 20 times on the smallest tour. The most promising mutation operators were RSM and HPRM,

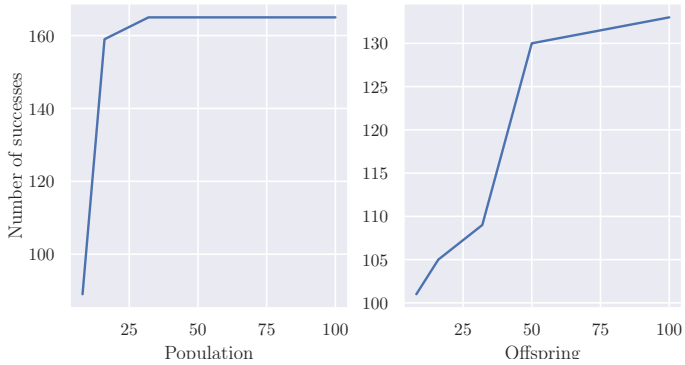

Figure 2: Number of successes for different parameter values on `tour29`.

Table 1: Parameter values for the hyperparameter search

Parameter	Values
λ	8, 16, 32, 100
μ	8, 16, 32, 50, 100
l	0, 0.25, 0.5, 0.75, 1

but there was no recombination operator that was significantly better. The remaining mutation operators and all recombination operators were then tested on `tour194.csv` in order to attempt to create a more significant difference between the recombination operators. The experiment showed that OX and SCX performed similar whilst PMX performed a fraction worse. For the final evolutionary algorithm OX was chosen, because it typically needed less generations than SCX to converge. Note that during this experiment it was assumed that all operators had similar performance with the same parameters. This is normally not the case.

Now that the operators are chosen, different experiments can be performed on the parameters of the evolutionary algorithm. If a value of a parameter was suggested by the authors of a paper, then this value was used and not changed or included in the hyperparameter search. The following parameters were selected to perform a hyperparameter search on: the population size λ, the offspring size μ and the percentage of individuals to perform local search on l. In Table 1 all the tested values for the different parameters can be found.
The hyperparameter search was performed on the smallest problem instance `tour29`. Each combination was tested eight times in order to reduce the random effect of the operators. Since the mean values were not relevant as indicator for the performance, the success rate was used. Where success is defined as finding the global minimum tour length. Obviously bigger population and offspring sizes show a higher success rate, as seen in Figure 2, but there ain't no such thing as a free lunch [13]. Therefore promising values that demand less calculations, like a population size around 25 and an offspring size around 50, are explored in a new hyperparameter search on a larger problem instance.

This larger problem instance is `tour194` and the parameters used are 16, 25 and 32 for the population size λ and 40, 50 and 60 for the offspring size μ. The search was not able to significantly differentiate between the parameters used. Therefore the values of these parameters can be chosen by personal preference. For the rest of this paper the values $\lambda = 16$ and $\mu = 50$ are used.

3.12 Other considerations

Some operators, like the local search operator and the fitness sharing elimination, are computationally expensive. But the results of these operators don't depend on each other. Therefore these operators can be executed in parallel. To accomplish this a `RawArray` from `multiprocessing` is used for the distance matrix. This is a one dimensional array that cannot be locked, but this is not needed because the parallel processes will not alter the matrix.

Sometimes during the execution of the algorithm, it occurs that it gets stuck in a local minimum and the diversity and local search operators are not able to get the individual out of it. To cope with this problem, the evolutionary algorithm 'restarts' itself by initializing a completely new population when the best individual has not been changed for a set number of iterations. This initialization is done with the random nearest insertion algorithm as described in section 3.2.

Elitism is used to stop the algorithm from diverging from the optimal value. If the best individual after elimination is worse than the best individual of the previous generation, then this individual is inserted and the current worst individual is discarded.

4 Numerical experiments

4.1 Metadata

All experiments conducted in this section are done using the same parameter values. The parameters that have a fixed value can be found in Table 2 and the dynamic parameters take the values specified in section 3. The operators used are: roulette wheel selection, OX for recombination, RSM for mutation, parallel 3-opt for local search and duplicate individual removal as diversity promotion scheme. All problem instances were run for five minutes, but the view of the graphs has been limited to 30 generations after the best objective value had been found. This is done to show the convergence more clearly.

All numeric experiments were performed on a 64-bit version of Ubuntu 20.04 with a quad-core Intel i5-7300U @ 2.60 GHz and 16 GB of memory. Python version 3.8.5 was used.

Table 2: Static parameters used in the numerical experiments

Parameter	Value	Parameter	Value
Population size λ	16	Offspring size μ	50
Number of nearest neighbors	15	Selection pressure	0.999
Margin σ in greedy diversification	0.10	Selection pressure decay	0.999

4.2 tour29.csv

In Figure 5 the convergence graph of this problem can be found in the top left hand corner. Since the heuristic value is a good approximation for the optimal solution, it converges quickly. Because of these large values a plot with only the convergence of the best objective value has been added in Figure 6.
The solution found is the global optimal solution for this instance. The corresponding sequence is:

$$14, 11, 10, 9, 5, 0, 1, 4, 7, 3, 2, 6, 8, 12, 13, 15, 23, 24, 26, 19, 25, 27, 28, 22, 21, 20, 16, 17, 18$$

It finds this solution very quickly. In Figure 6, it can clearly be seen that the optimal solution is found around generation 25, after approximately 2 seconds. It is also very memory efficient since all the operations happen in place. In Figure 3 the results of the memory profiling can be seen for the duration of the algorithm. The blue brackets mark the start and the end of the optimize function call. Since the results of the bigger tours show similar results, apart from the height of the plateau, these graphs will be omitted.
The algorithm will generate a different solution each time it is executed due to the random effects present. To study the variations in the solutions, the problem has been solved 1000 times. A histogram of the mean and best objective values is shown in Figure 4. Note that the left-most bin of the best objective value extends up to 890, but the view of the plot has been restricted to 80 in order to make the other bins visible. The averages of the mean and best values are 29737 and 27206 respectively, while the standard deviations are 731 and 157.

4.3 tour100.csv

Just like the convergence graph of `tour29`, the convergence graph of `tour100` can be found on Figure 5 and a clearer view of the best value on is shown Figure 6. Note that the mean objective value shows some incontinuities due to the disconnected cities. The algorithm does not explicitly take these disconnections into account, but rather treats it as an edge with an infinitely high distance. Therefore the recombination and mutation operators are able to create tours with an infinite length. This does not pose a problem for the best solution since the elimination and selection schemes will filter these results out.
The solution found is only 1% longer than the optimal value specified in the project description and this optimal value is an estimation in itself. So the possibility exists that the solution found is even more optimal. The solution is also found well within the time limit (after around 30 seconds).

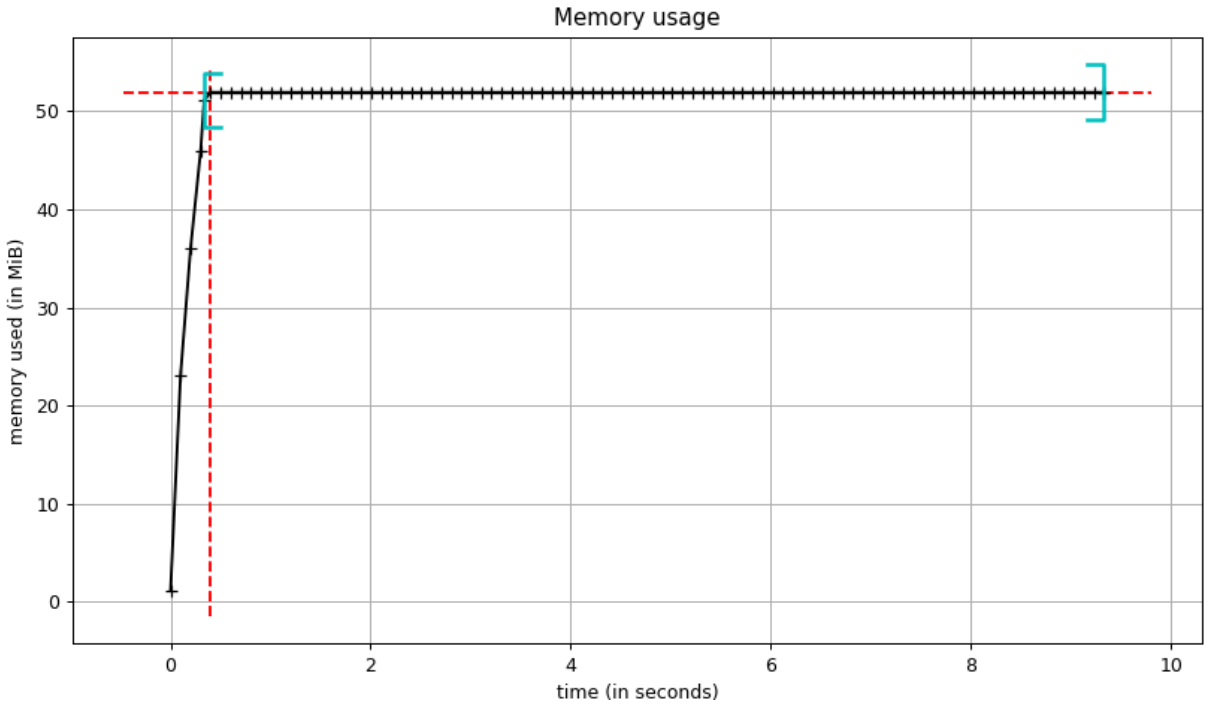

Figure 3: Memory profiling of the `optimize` routine for `tour29`.

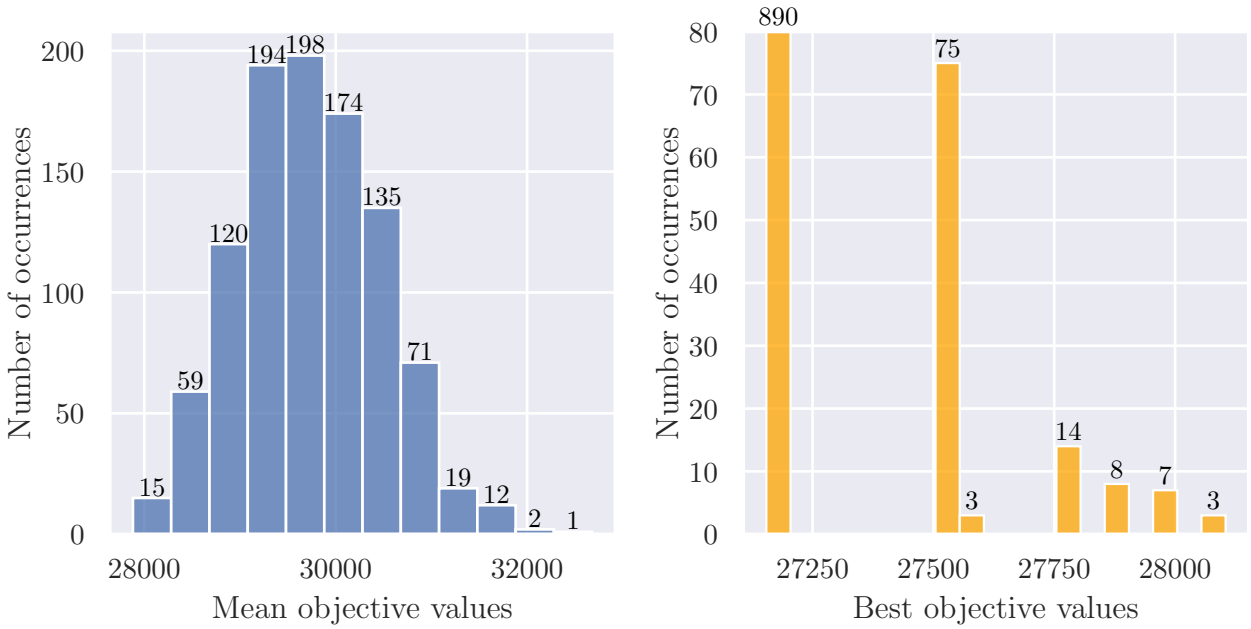

Figure 4: Histogram of the mean and best objective values of 1000 runs of the `tour29` problem instance.

4.4 tour194.csv

Just like the convergence graph of `tour29`, the convergence graph of `tour194` can be found on Figure 5 and a clearer view of the best value on is shown Figure 6. The various restart points are clearly visible in the mean objective value. This time the algorithm ran for the full time span given, but the final solution was already found after approximately 100 seconds. The solution found here belongs in the top 20% of all students on the leaderboard (at the time of writing). This shows that the solution found is a good solution for this problem.

4.5 tour929.csv

Just like the convergence graph of `tour29`, the convergence graph of `tour929` can be found on Figure 5 and a clearer view of the best value is shown on Figure 6. The final best solution had a length of 102637.963, but the algorithm had not yet converged after five minutes and 135 generations. This solution is 7% longer than the optimal value specified in the project description. But this is again an estimation, thus the difference can be smaller as well. The best value this algorithm was able to produce, when no time limit was applied, was 100810 in 544 generations and 27 minutes.

5 Critical reflection

This project was very useful to get a better understanding of the workings of evolutionary algorithms. I knew that these algorithms existed and had seen them applied to games in different YouTube videos [14, 15], but the fact that they can be applied to other problems as well was new to me.
I think that one of the key strengths of evolutionary algorithms is the fact that the search space can be explored in a structured way instead of sampling the domain at random and hoping for a good solution. But this strength can be its downfall as well. If the algorithm finds a local optimum, it can get stuck in this optimum. This weakness is further increased by the use of local search operators. Luckily diversion promotion schemes exist that try to counteract these weaknesses.
All these operators are used in the algorithm described in this paper. The biggest weakness of this algorithm is the local search operator. It is very powerful and this sometimes causes the algorithm to converge to local optima. Different diversity promotion schemes were tested to handle this problem but none of them were ideal. Some were computationally too expensive to perform in the five minute time span, whilst others were too disruptive. This caused that the algorithm essentially needed to converge twice in the five minutes given. A less powerful local search operator could mitigate the problem, but then more time is needed in order to converge. I think that the algorithm I have implemented is a good compromise between computation and results.
Another weakness of evolutionary algorithms is the need for parameter tuning. This surprised me, because most of the times the results of the parameters 'out-of-the-box' are significantly worse than the tuned parameters. I did not know there were so many parameters to choose. Self-adaptivity can help with this problem, but sometimes these schemes require starting parameters themselves. In this case the parameter selection took almost 30 times longer than the algorithm is allowed to run. It would almost be worth using an evolutionary algorithm to set the parameters of the evolutionary algorithm.
I will definitely try out evolutionary algorithms on other problems in the future. The capabilities are almost endless with the different mutation, recombination and other operators.

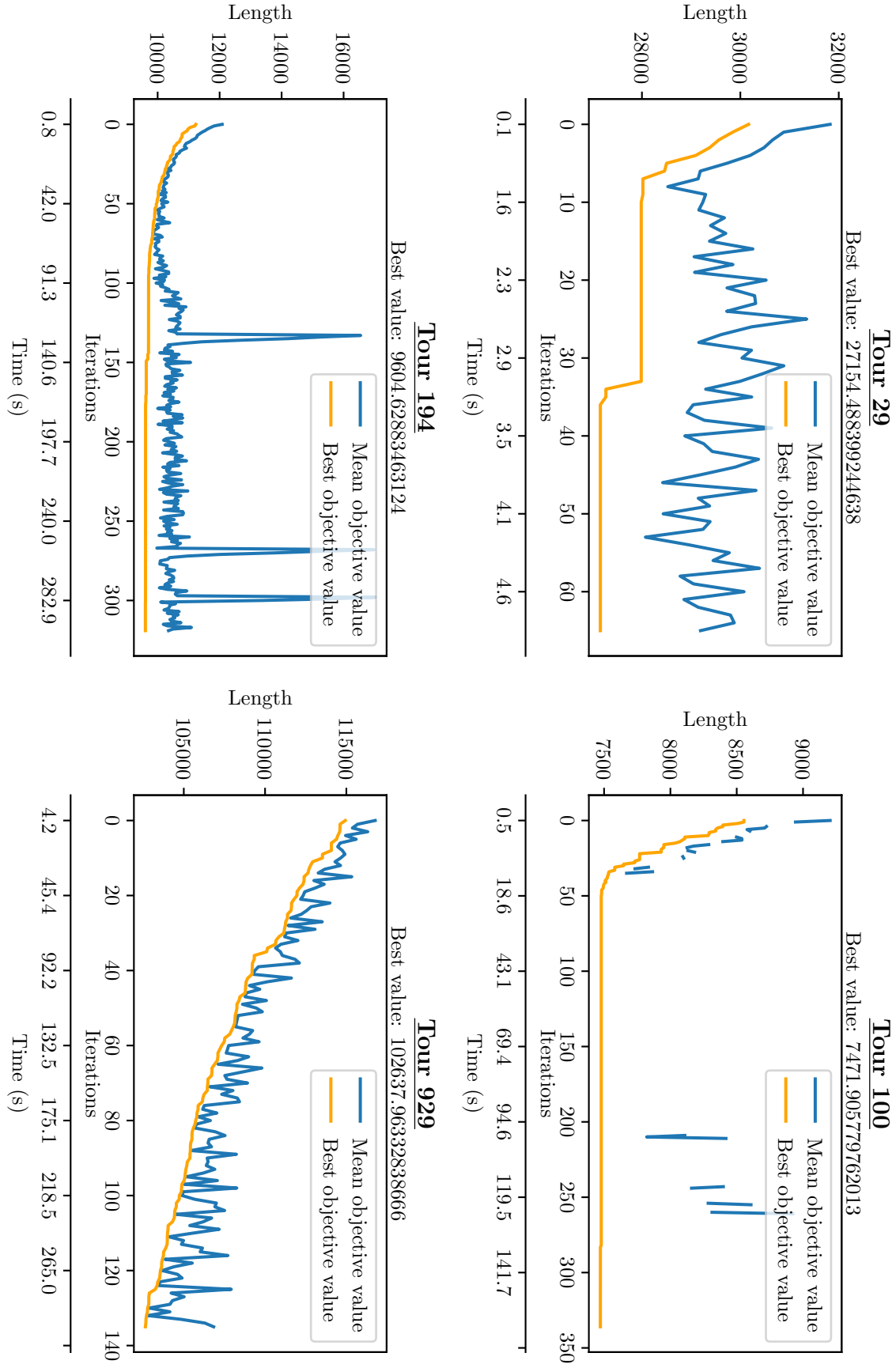

Figure 5: Convergence graphs of the different benchmark tours.

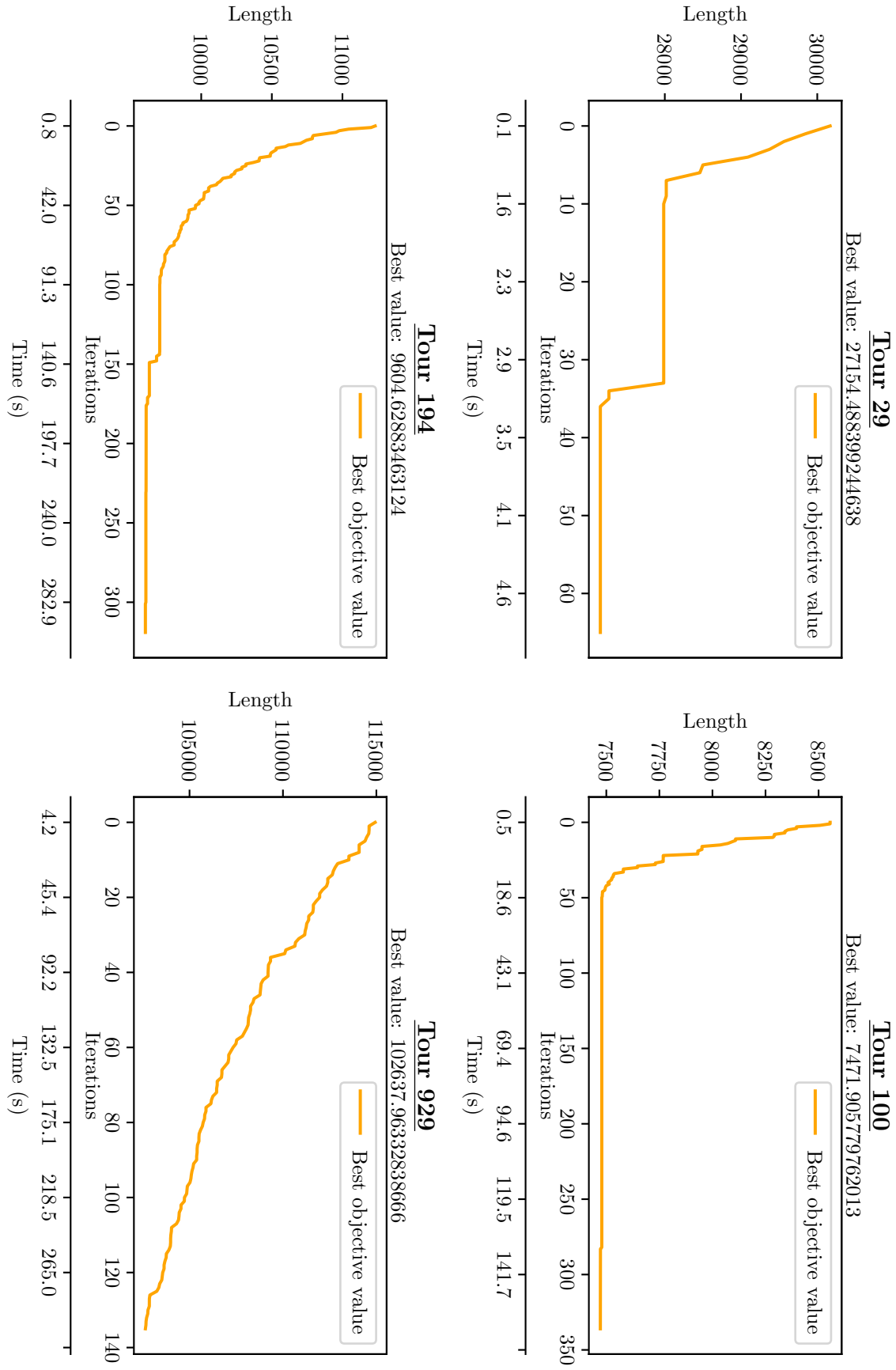

Figure 6: Convergence graphs of the best objective values of the different benchmark tours.

References

[1] Jihene Kaabi and Youssef Harrath. Permutation rules and genetic algorithm to solve the traveling salesman problem. *Arab Journal of Basic and Applied Sciences*, 26(1):283–291, 2019.

[2] Y Sakurai, K Takada, N Tsukamoto, T Onoyama, R Knauf, and S Tsuruta. Inner random restart genetic algorithm to optimize delivery schedule. In *2010 IEEE International Conference on Systems, Man and Cybernetics*, pages 263–270. IEEE, 2010.

[3] Keith Schwarz. Darts, dice, and coins: Sampling from a discrete distribution. `https://www.keithschwarz.com/darts-dice-coins`, 2011. Accessed: 2020-11-13.

[4] Michael D. Vose. A linear algorithm for generating random numbers with a given distribution. *Software Engineering, IEEE Transactions on*, 1991.

[5] Nick Vannieuwenhoven. Genetic algorithms and evolutionary computing, 2020.

[6] Abdoun Otman, Chakir Tajani, and Jaafar Abouchabka. A new mutation operator for solving an np-complete problem: Travelling salesman problem. 05 2012.

[7] Lawrence Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'85, page 162–164, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

[8] Zakir Ahmed. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometric and Bioinformatics*, 3, 03 2010.

[9] Paris-C. Kanellakis and Christos H. Papadimitriou. Local search for the asymmetric traveling salesman problem. *Operations Research*, 28(5):1086–1099, 1980.

[10] Jon Louis Bentley. Experiments on traveling salesman heuristics. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, page 91–99, USA, 1990. Society for Industrial and Applied Mathematics.

[11] Yuichi Nagata and David Soler. A new genetic algorithm for the asymmetric traveling salesman problem, expert systems with applications. *Expert Systems with Applications*, 39(10):8947–8953, 2012.

[12] Andrés Herrera-Poyatos and Francisco Herrera. Genetic and memetic algorithm with diversity equilibrium based on greedy diversification. *CoRR*, abs/1702.03594, 2017.

[13] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[14] Code Bullet. Ai learns to play snake using genetic algorithm and deep learning. `https://www.youtube.com/watch?v=3bhP7zulFfY`, 12 2017.

[15] Code Bullet. Using a.i. to dominate nerds in tetris. `https://www.youtube.com/watch?v=os4DcbpL0Nc`, 09 2020.